类型系统初识

学习目标

- 了解类型系统
  - 类型标注
  - 类型检测的好处
  - 使用场景
- 掌握常用的类型标注的使用

什么是类型

程序 = 数据结构 + 算法 = 各种格式的数据 + 处理数据的逻辑

数据是有格式（类型）的

- 数字、布尔值、字符
- 数组、集合

程序是可能有错误的

- 计算错误（对非数字类型数据进行一些数学运算）
- 调用一个不存在的方法

> 不同类型的数据有不同的操作方式或方法，如：字符串类型的数据就不应该直接参与数学运算

动态类型语言 & 静态类型语言

动态类型语言

程序运行期间才做数据类型检查的语言，如：JavaScript

静态类型语言

程序编译期间做数据类型检查的语言，如：Java

静态类型语言的优缺点

优点

- 程序编译阶段（配合IDE、编辑器甚至可以在编码阶段）即可发现一些潜在错误，避免程序在生产环境运行了以后再出现错误
- 编码规范、有利于团队开发协作、也更有利于大型项目开发、项目重构
- 配合IDE、编辑器提供更强大的代码智能提示/检查
- 代码即文档

缺点

- 麻烦
- 缺少灵活性

动态类型语言

优点

- 静态类型语言的缺点

缺点

- 静态类型语言的优点

静态类型语言的核心： 类型系统

什么是类型系统

类型系统包含两个重要组成部分

- 类型标注（定义、注解）- typing
- 类型检测（检查）- type-checking

类型标注

类型标注就是在代码中给数据（变量、函数（参数、返回值））添加类型说明，当一个变量或者函数（参数）等被标注以后就不能存储或传入与标注类型不符合的类型

有了标注，`TypeScript` 编译器就能按照标注对这些数据进行类型合法检测。

有了标注，各种编辑器、IDE等就能进行智能提示

类型检测

顾名思义，就是对数据的类型进行检测。注意这里，重点是类型两字。

类型系统检测的是类型，不是具体值（虽然，某些时候也可以检测值），比如某个参数的取值范围（1-100之间），我们不能依靠类型系统来完成这个检测，它应该是我们的业务层具体逻辑，类型系统检测的是它的值类型是否为数字！

类型标注

在 `TypeScript` 中，类型标注的基本语法格式为：

```
数据载体:类型
```

`TypeScript` 的类型标注，我们可以分为

- 基础的简单的类型标注
- 高级的深入的类型标注

基础的简单的类型标注

- 基础类型
- 空和未定义类型
- 对象类型
- 数组类型
- 元组类型
- 枚举类型
- 无值类型
- Never类型
- 任意类型
- 未知类型（Version3.0 Added）

基础类型

基础类型包含：string，number，boolean

标注语法

```
let title: string = '开课吧';
let n: number = 100;
let isOk: boolean = true;
```

空和未定义类型

因为在 `Null` 和 `Undefined` 这两种类型有且只有一个值，在标注一个变量为 `Null` 和 `Undefined` 类型，那就表示该变量不能修改了

```
let a: null;
// ok
a = null;
// error
a = 1;
```

默认情况下 `null` 和 `undefined` 是所有类型的子类型。 就是说你可以把 `null` 和 `undefined` 其它类型的变量

```
let a: number;
// ok
a = null;
```

如果一个变量声明了，但是未赋值，那么该变量的值为 `undefined`，但是如果它同时也没有标注类型的话，默认类型为 `any`，`any` 类型后面有详细说明

```
// 类型为 `number`，值为 `undefined`
let a: number;
// 类型为 `any`，值为 `undefined`
```

小技巧

因为 `null` 和 `undefined` 都是其它类型的子类型，所以默认情况下会有一些隐藏的问题

```
let a:number;
a = null;
// ok（实际运行是有问题的）
a.toFixed(1);
```

> 小技巧：指定 `strictNullChecks` 配置为 `true`，可以有效的检测 `null` 或者 `undefined`，避免很多常见问题

```
let a:number;
a = null;
// error
a.toFixed(1);
```

> 也可以使我们程序编写更加严谨

```
let ele = document.querySelector('div');
// 获取元素的方法返回的类型可能会包含 null，所以最好是先进行必要的判断，再进行操作
if (ele) {
        ele.style.display = 'none';
}
```

对象类型

内置对象类型

在 `JavaScript` 中，有许多的内置对象，比如：Object、Array、Date.....，我们可以通过对象的 构造函数 或者 类 来进行标注

```
let a: object = {};
// 数组这里标注格式有点不太一样，后面我们在数组标注中进行详细讲解
let arr: Array<number> = [1,2,3];
let d1: Date = new Date();
```

自定义对象类型

另外一种情况，许多时候，我们可能需要自定义结构的对象。这个时候，我们可以：

- 字面量标注
- 接口
- 定义 类 或者 构造函数

字面量标注：

```
let a: {username: string; age: number} = {
  username: 'zMouse',
  age: 35
};
// ok
a.username;
a.age;
// error
a.gender;
```

优点：方便、直接

缺点：不利于复用和维护

接口：

```
// 这里使用了 interface 关键字，在后面的接口章节中会详细讲解
interface Person {
  username: string;
  age: number;
};
let a: Person = {
  username: 'zMouse',
  age: 35
};
// ok
a.username;
a.age;
// error
a.gender;
```

优点：复用性高

缺点：接口只能作为类型标注使用，不能作为具体值，它只是一种抽象的结构定义，并不是实体，没有具体功能实现

类与构造函数：

```
// 类的具体使用，也会在后面的章节中讲解
class Person {
    constructor(public username: string, public age: number) {
  }
}
// ok
a.username;
a.age;
// error
a.gender;
```

`优点`：功能相对强大，定义实体的同时也定义了对应的类型

`缺点`：复杂，比如只想约束某个函数接收的参数结构，没有必要去定一个类，使用接口会更加简单

```
interface AjaxOptions {
    url: string;
    method: string;
}

function ajax(options: AjaxOptions) {}

ajax({
    url: '',
    method: 'get'
});
```

扩展

包装对象：

这里说的包装对象其实就是 `JavaScript` 中的 `String`、`Number`、`Boolean`，我们知道 `string` 类型 和 `String` 类型并不一样，在 `TypeScript` 中也是一样

```
let a: string;
a = '1';
// error String有的，string不一定有（对象有的，基础类型不一定有）
a = new String('1');

let b: String;
b = new String('2');
// ok 和上面正好相反
b = '2';
```

数组类型

`TypeScript` 中数组存储的类型必须一致，所以在标注数组类型的时候，同时要标注数组中存储的数据类型

使用泛型标注

```
// <number> 表示数组中存储的数据类型，泛型具体概念后续会讲
let arr1: Array<number> = [];
// ok
arr1.push(100);
// error
arr1.push('开课吧');
```

简单标注

```
let arr2: string[] = [];
// ok
arr2.push('开课吧');
// error
arr2.push(1);
```

元组类型

元组类似数组，但是存储的元素类型不必相同，但是需要注意：

- 初始化数据的个数以及对应位置标注类型必须一致
- 越界数据必须是元组标注中的类型之一（标注越界数据可以不用对应顺序 - 联合类型）

```
let data1: [string, number] = ['开课吧', 100];
// ok
data1.push(100);
// ok
data1.push('100');
// error
data1.push(true);
```

枚举类型

枚举的作用组织收集一组关联数据的方式，通过枚举我们可以给一组有关联意义的数据赋予一些友好的名字

```
enum HTTP_CODE {
  OK = 200,
  NOT_FOUND = 404,
  METHOD_NOT_ALLOWED
};
// 200
HTTP_CODE.OK;
// 405
HTTP_CODE.METHOD_NOT_ALLOWED;
// error
HTTP_CODE.OK = 1;
```

注意事项：

- key 不能是数字
- value 可以是数字，称为 数字类型枚举，也可以是字符串，称为 字符串类型枚举，但不能是其它值，默认为数字：0
- 枚举值可以省略，如果省略，则：
  - 第一个枚举值默认为：0
  - 非第一个枚举值为上一个数字枚举值 + 1
- 枚举值为只读（常量），初始化后不可修改

字符串类型枚举

枚举类型的值，也可以是字符串类型

```
enum URLS  {
  USER_REGISETER = '/user/register',
  USER_LOGIN = '/user/login',
  // 如果前一个枚举值类型为字符串，则后续枚举项必须手动赋值
  INDEX = 0
}
```

注意：如果前一个枚举值类型为字符串，则后续枚举项必须手动赋值

> 小技巧：枚举名称可以是大写，也可以是小写，推荐使用全大写（通常使用全大写的命名方式来标注值为常量）

无值类型

表示没有任何数据的类型，通常用于标注无返回值函数的返回值类型，函数默认标注类型为：`void`

```
function fn():void {
    // 没有 return 或者 return undefined
}
```

> 在 `strictNullChecks` 为 `false` 的情况下，`undefined` 和 `null` 都可以赋值给 `void` ，但是当 `strictNullChecks` 为 `true` 的情况下，只有 `undefined` 才可以赋值给 `void`

Never类型

当一个函数永远不可能执行 `return` 的时候，返回的就是 `never` ，与 void 不同，`void` 是执行了 `return` ，只是没有值，`never` 是不会执行 `return` ，比如抛出错误，导致函数终止执行

```
function fn(): never {
    throw new Error('error');
}
```

任意类型

有的时候，我们并不确定这个值到底是什么类型或者不需要对该值进行类型检测，就可以标注为 `any` 类型

```
let a: any;
```

- 一个变量申明未赋值且未标注类型的情况下，默认为 `any` 类型
- 任何类型值都可以赋值给 `any` 类型
- `any` 类型也可以赋值给任意类型
- `any` 类型有任意属性和方法

注意：标注为 `any` 类型，也意味着放弃对该值的类型检测，同时放弃 IDE 的智能提示

> 小技巧：当指定 `noImplicitAny` 配置为 `true` ，当函数参数出现隐含的 `any` 类型时报错

未知类型

unknow，3.0 版本中新增，属于安全版的 any，但是与 any 不同的是：

- unknow 仅能赋值给 unknow、any
- unknow 没有任何属性和方法

函数类型

在 JavaScript 函数是非常重要的，在 TypeScript 也是如此。同样的，函数也有自己的类型标注格式

- 参数
- 返回值

```
函数名称( 参数1: 类型, 参数2: 类型... ): 返回值类型;
```

```
function add(x: number, y: number): number {
    return x + y;
}
```

函数更多的细节内容，在后期有专门的章节来进行深入的探讨